Safa Slote
CS451: Parallel and Distributed Computing
05/05/2021

CUDA Matrix Normalization

Machine Information

CC-CentOS7-CUDA10-2003

Discussion

Description of Algorithm

Starting with the main function, we first start by initializing all scalability variables. We then call the parameters function and initialize the inputs. Both of these functions have tweaks in them so that they are more CUDA compatible. Instead of utilizing a two-dimensional array for my matrices, I had to convert them into one-dimensional arrays so that I can assign a pointer to them in order to use them for the CUDA memcpy functions. Because I converted matrix A and B to one-dimensional arrays, I had to index them a different way, in which I used col*N+row to access the elements in the iterations. I them start the clock and use functions gettimeofday() and times() to compute start of cpu times. I declare two float pointers assigned to matrix A and B to pass onto cuda Malloc and Memcpy in order to make changes to the matrices and return them back. I cuda malloc two float pointers that are destination pointers with the source pointers. I then use the cudaEvent method and declare start and stop timers. I then use Memcpy the pointers from host to device (CPU to GPU) in order to do work in shared memory and eventually spit it back into global memory. I then use the function Cuda event record to start the timer and call the function matrixNorm, that takes on the parameters number of blocks and number of threads from the user.

In matrixNorm, I initialized the rows and columns as dimensions of the grids with thread ID's. The algorithm is the same as the sequential code except the function __syncthreads() are placed before places that need to be completed. For example, __syncthreads() are placed before sigma because the mean absolutely needs to be calculated before you calculate sigma. __syncthreads() is placed before the last iteration because sigma must be calculated before that iteration in order to calculate the normalized function. After calling the function, I cudaFree() whatever memory there was and calculated timestamps.

I also edited the sequential code matrixNorm.c and included an argument that took number of threads along with time stamps.

Iterations
The iterations are kept the same from the sequential code except for the indexing. Because I had to map two-dimensional arrays into one, I had to use index functions [col *n + row] and [row*n+col] to access the array elements. The first and second iteration calculates the mean, third iteration calculates sigma using the mean, then third iteration normalizes the matrix and puts in it B which we use to send the device back to the host to get our final answer.

Efficiency
This code is efficient because it uses synchronization techniques that ensures all threads are in sync before moving onto the algorithm. It is also efficient because it saves time making threads that take on different instructions instead of having a sequential code that takes a long time getting through the code.

Correctness
The code compiles with no errors and does print the timestamps that I asked for.

Scalability
Comparing CPU performance to GPU performance
From the results, we can clearly see that the GPU is performing much better than the CPU (sequential code)

<u>Sequential</u>
N = 3000
Random Seed: 5
Number of Threads: 5
Elapsed Time: 1.84451e+16 ms
CPU Time: 2.80823e+11 ms

N = 3000
Random Seed = 5
Number of Threads = 10
Elapsed Time = 1.84451e+16 ms
CPU Time = 2.8106e+11 ms

GPU

N = 3000

Random Seed = 5

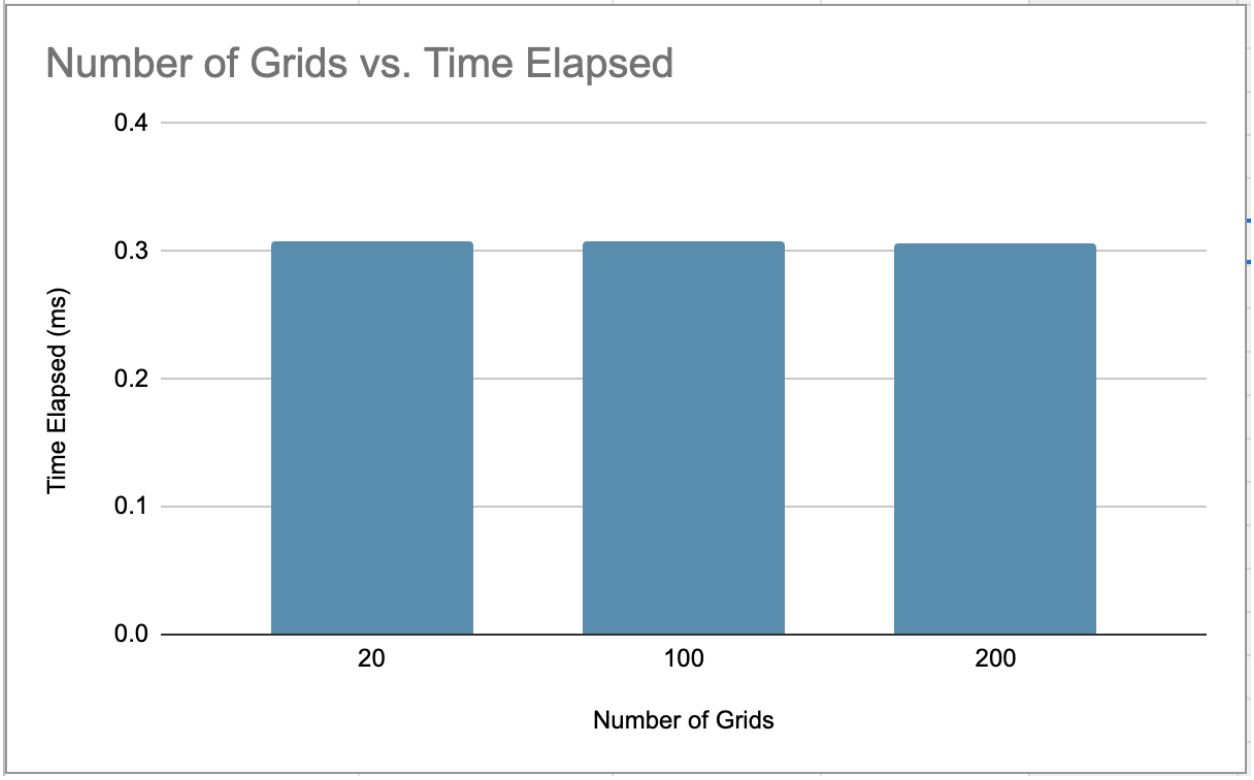
Number of Threads = 7

Number of Blocks = 1

Elapsed Time = 3140.46 ms

CPU Time = 0.299 ms

Number of Threads	CPU
20	0.308
100	0.307
200	0.306



N = 3000

Random Seed = 5

Number of Threads = 10

Number of Blocks = 1

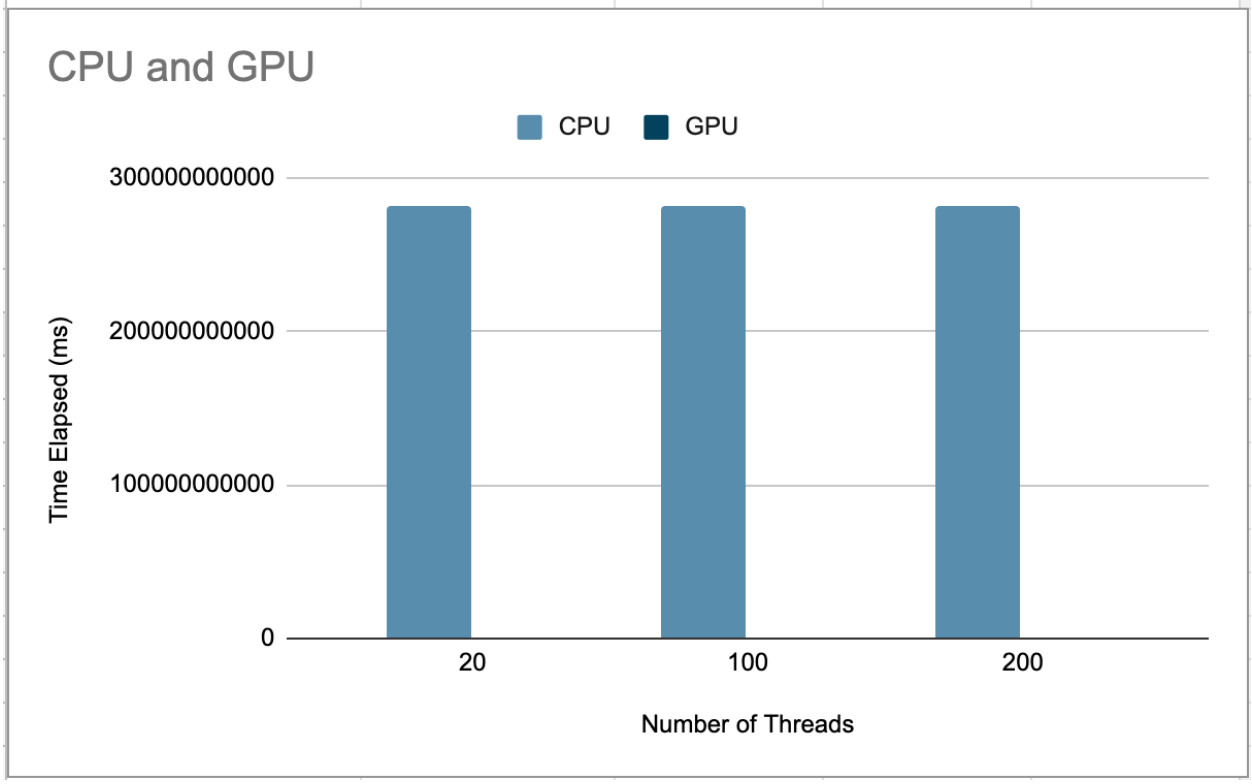
Elapsed Time = 3165.71 ms
CPU Time = 0.302 ms

N = 3000
Random Seed = 5
Number of Threads = 100
Number of Blocks = 10
Elapsed Time = 3189.62 ms
CPU Time = 0.295 ms

Based on the chart, we see that there is a huge difference in time elapsed in both of them. GPU clearly performs better than CPU.

I also measured in GPU the how time elapsed would change as I increase the number of grids. Here are my results:

Number of Threads	CPU	GPU		
20	281000000000	0.301		
100	281020000000	0.303		
200	281413000000	0.308		



The time decreased slightly but there was still better performance in increasing the number of grids